An Introduction to FreeBSD 6 Kernel Hacking

Lawrence Stewart, James Healy
Centre for Advanced Internet Architectures, Technical Report 070622A
Swinburne University of Technology
Melbourne, Australia
lastewart@swin.edu.au, jhealy@swin.edu.au

***Abstract*—The widely used FreeBSD UNIX-like operating system provides a mature, stable and customisable platform, suitable for many tasks including telecommunications research. FreeBSD is being used as part of CAIA's NewTCP project [1] to evaluate up and coming TCP congestion control algorithms. Part of this work has involved the customisation of the FreeBSD kernel, in the form of a loadable kernel module named SIFTR (Statistical Information For TCP Research). This report aims to capture the knowledge learnt during this process. Whilst FreeBSD 6.2-RELEASE was used as the basis for this development, most of the technical information in this report will be applicable to all FreeBSD 6.x releases, and possibly to earlier (5.x and 4.x to a lesser extent) or up and coming (7.x and beyond) releases. Topics covered include FreeBSD kernel module programming, the sysctl interface, packet filtering, data structures, character devices, threading, file writing and debugging. The report draws together our personal experiences and sources of further information to create a useful reference for those that are new to FreeBSD kernel programming.**

***Index Terms*—FreeBSD, Kernel, Hacking, Programming, Module, NewTCP**

I. Introduction

This technical report discusses many tips, tricks and lessons learnt during the development of a FreeBSD kernel module named SIFTR (Statistical Information For TCP Research) [2] for the NewTCP [1] project at CAIA. Whilst FreeBSD 6.2-RELEASE was used as the basis for this development, most of the technical information in this report will be applicable to all FreeBSD 6.x releases, and possibly to earlier (5.x and 4.x to a lesser extent) or up and coming (7.x and beyond) releases.

Over the course of developing the SIFTR kernel module, a great deal of useful and in some cases hard to find information was learnt about programming in the FreeBSD kernel environment. This report should act as a primer to understanding the FreeBSD kernel architecture, environment and mechanisms useful in the development of FreeBSD kernel code. We also try to provide many useful avenues for finding further information about the topics discussed in the report. We will be making specific reference to code snippets taken from the SIFTR code, which is available for download from [2] and should be referred to whilst reading this report.

FreeBSD is a well known and widely used open source UNIX-like operating system. It is known as an extremely stable and mature platform that has been developed and fine tuned over the past 15 years.

Modifications of kernel level functions, such as the instrumentation we required for SIFTR, are often done by creating a "patch" containing the required changes against the kernel's source code. Whilst relatively easy to apply, such patches are irritating to develop and deploy because of the need to modify kernel sources directly and recompile the kernel each time the patch changes. Thankfully, there are nicer ways of changing kernel behaviour.

FreeBSD's dynamic kernel linker (KLD) facility [3] allows code modules to be dynamically loaded into and unloaded from a running FreeBSD kernel. This allows system users with appropriate privileges to modify the kernel's runtime behaviour very quickly and easily by issuing a single line command to load the new module. Kernel modules also make life easier for the developer, as only the kernel module itself needs to be recompiled each time a change is made.

II. Anatomy of a Kernel Module

Every kernel module starts with a call to the DECLARE_MODULE() macro (defined in /usr/src/sys/sys/module.h), which defines, in order, the name, code execution entry point, category and load order for the module. Listing 1 shows the call to DECLARE_MODULE() from the SIFTR code. The siftr_mod argument is a moduledata_t struct, which contains the pointer to the code execution entry point (in our case a function named "load_handler").

Listing 1

```
DECLARE_MODULE(siftr, siftr_mod, SI_SUB_KLD,
SI_ORDER_ANY);
```

The code execution entry point is called each time the module is loaded and unloaded. This allows the module to perform any necessary initialisation on start up and cleanup on shutdown as required.

Listing 2 is a sample Makefile that simplifies the building of a KLD. The bsd.kmod.mk file is provided as part of the FreeBSD distribution and a significant amount of work is saved by including it. Its behaviour is controlled by a range of environment variables, three of which are shown. A simple usage example is available at [4].

Listing 2

```
SRCS=vnode_if.h subr_kernio.c siftr.c
KMOD=siftr
CFLAGS=-g -O3
.include <bsd.kmod.mk>
```

Having created your Makefile and source code, you can simply run "make" from the command line and then "kldload ./siftr.ko" to load your module into the running kernel. The "./" infront of your module name is important as by default, kldload will look for kernel modules in /boot/kernel and /boot/modules, and will not find your kernel module in the local directory.

III. Sysctl Variables

The sysctl system interface [5] provides a means for user-space processes to query and manipulate kernel-space state information. It therefore provides an ideal way to communicate with our kernel module to change its configuration parameters.

Sysctl variables are arranged in a tree like hierarchy, with variables hanging off branches of the tree e.g. net.inet.ip.ttl, where net.inet.ip represents the branch of the sysctl tree and ttl is the actual variable. Sysctl variables are always found as leaves on the tree, and a branch cannot also be used as a variable e.g. in the previous example, net.inet.ip could not refer to a variable, but can have other branches and variables hanging off it. This structure provides a very simple way to organise and group system variables.

The SIFTR module exports 3 configuration paramaters as read/write sysctl variables (you can make variables read-only or write-only). These variables reside under the net.inet.siftr sysctl branch.

Listing 3 is used to declare the net.inet.siftr sysctl container struct.

Listing 3

```
SYSCTL_DECL(_net_inet_siftr);
```

Listing 4 declares a sysctl node "siftr" which will hang off the net.inet branch of the sysctl tree. Node's act as branches within the sysctl tree, and cannot be used as variables in their own right.

Listing 4

```
SYSCTL_NODE(_net_inet, OID_AUTO, siftr,
CTLFLAG_RW, NULL, "siftr related settings");
```

Listing 5 declares a read/write sysctl variable named "enabled" of type unsigned int, which will hang off the net.inet.siftr branch of the sysctl tree. This variable is used to control whether the module's functionality is switched on or off when loaded. The initial value of the variable is set to 0, and when the variable is changed, the function siftr_sysctl_enabled_handler() will be called to handle the change. The variable's data storage is provided by the variable named "siftr_enabled".

Listing 5

```
SYSCTL_OID(_net_inet_siftr, OID_AUTO, enabled,
CTLTYPE_UINT|CTLFLAG_RW, &siftr_enabled, 0,
&siftr_sysctl_enabled_handler, "IU", "switch siftr mod-
ule operations on/off");
```

Listing 6 declares a read/write sysctl variable named "ppl" of type unsigned int, which will hang off the net.inet.siftr branch of the sysctl tree. This variable is used to control how many observed packets for a particular TCP connection will trigger a log message to be written. The initial value of the variable is set to 1 i.e. write a log message for every packet per connection. When the variable is changed, the function siftr_sysctl_pkts_per_log_handler() will be called to handle the change. The variable's data storage is provided by the variable named "siftr_pkts_per_log".

Finally, listing 7 declares a read/write sysctl variable named "logfile" of type string, which will hang off the net.inet.siftr branch of the sysctl tree. This variable is used to specify the path to the file that SIFTR log messages will be written to. When the variable is changed,

Listing 6

```
SYSCTL_OID(_net_inet_siftr, OID_AUTO, ppl, CTLTYPE_UINT|CTLFLAG_RW, &siftr_pkts_per_log, 1, &siftr_sysctl_pkts_per_log_handler, "IU", "number of packets between generating a log message");
```

the function siftr_sysctl_logfile_name_handler() will be called to handle the change. The variable's data storage is provided by the variable named "siftr_logfile".

Listing 7

```
SYSCTL_PROC(_net_inet_siftr, OID_AUTO, logfile, CTLTYPE_STRING|CTLFLAG_RW, &siftr_logfile, sizeof(siftr_logfile), &siftr_sysctl_logfile_name_handler, "A", "file to save siftr log messages to");
```

The end result of including the above code snippets in the module is that *net.inet.siftr.enabled*, *net.inet.siftr.ppl* and *net.inet.siftr.logfile* will be made available as soon as the module is loaded, and removed when the module is unloaded.

There is an alternate technique for registering sysctl variables that works dynamically at runtime, allowing them to be added and removed as required while the module is running. Although we didn't require this functionality for SIFTR, it may be useful in certain cases. Refer to "man 9 sysctl_ctx_init" and "man 9 sysctl_add_oid" for information on how to dynamically create and destroy sysctl nodes and variables.

More information regarding sysctl can be found by reading the various sysctl man pages, namely "man 8 sysctl", "man 9 sysctl" and "man 9 sysctl_add_oid". The sysctl C header file, located at /usr/src/sys/sys/sysctl.h, is also a useful reference.

IV. PFIL Hooks

The PFIL [6] [7] kernel programming interface (KPI) provides a way of accessing packets entering and leaving the network stack. It allows a developer to insert their own functions (which is known as "hooking") into the network stack at appropriate points to intercept packets. Depending on the return value of the function, the packet can be allowed to continue through the stack, or denied and dropped. PFIL is obviously very well suited to tasks like firewalling and packet manipulation. It also provides an ideal way for SIFTR to access TCP packets traversing the network stack and collect statistics about them and the connections they belong to.

The siftr_hook_pfil() function performs all the necessary actions to install and remove PFIL hooks. Listing 8 gets the head of the linked list of PFIL hooks currently installed for AF_INET (IPv4) packets.

Listing 8

```
pfh_inet = pfil_head_get(PFIL_TYPE_AF, AF_INET);
```

Then, depending on whether we are hooking in or unhooking, a call to pfil_add_hook() or pfil_remove_hook() respectively is made. These functions allow us to install or remove our custom hook function siftr_chkpkt() for both inbound and outbound IPv4 packets. Listing 9 shows how to install the siftr_chkpkt() hook function for both inbound and outbound packets. The second parameter to these functions can be used to pass an arbitrary pointer to the hook function each time it is invoked, but we did not require it and so left it as NULL.

Listing 9

```
pfil_add_hook(siftr_chkpkt, NULL, PFIL_IN | PFIL_OUT | PFIL_WAITOK, pfh_inet);
```

Hook functions suitable for use with PFIL must have a prototype as defined in the PFIL man page [6]. At the time of writing, the prototype is as shown in Listing 10. The function must return an int (0 indicating the packet is allowed to continue through the network stack, nonzero indicating the packet is to be blocked). It must take 5 arguments of the same type and in the same order as defined in Listing 10. The "void *arg" parameter will be populated with the value passed in as the second argument to the pfil_add_hook() function.

Listing 10

```
int (*func)(void *arg, struct mbuf **mp, struct ifnet *, int dir, struct inpcb *);
```

There are alternative ways to hook into the network stack e.g. by manipulating the protocol switch structure that defines the packet handling functions for each layer in the stack. Some examples are explored in the following references [8] [9].

V. Data Structures

Data structures are an integral part of any software, and knowing which ones to use and how to use them properly is a key part of writing high quality code. Userspace programming has the advantage of vast libraries

of data structures that are freely available to pick and choose from. Kernel-space programming on the other hand mandates the use of kernel-space data structures and if you don't like the ones provided, you have to write your own.

Lists, queues and hash tables are 3 of the most commonly used data structures, and fortunately, the kernel provides simplified functions to create and use all of them.

A. *Lists and Queues*

We will discuss lists and queues together, because a queue is simply implemented as a list that you insert elements at the back of and pull elements from the front of.

The list/queue KPI [10] defines a set of C macros used for simplifying the creation and manipulation of list/queue data structures. Although the code listings used here show the macros for working with tail queues, the macros for working with other types of queues and lists are very similar. SIFTR utilises a queue as a means of passing packets between the thread that runs the PFIL hook function and the packet manager thread.

Listing 11 defines a new struct named pkt_node, which acts as the fundamental unit of data stored in our queue. Each pkt_node in the queue will contain statistics and information relevant to the packet being processed, and a link to the next node in the list. The list is constructed by linking pkt_node structs together by pointing the pkt_node pointer of one pkt_node at the next pkt_node in the list.

Listing 12 defines a struct named pkt_queue which will act as the head of our queue. It contains a pointer to the first and last node in the list, which can be accessed using the STAILQ_FIRST() and STAILQ_LAST() macros respectively.

Listing 11

```
struct pkt_node
{
struct timeval tval;
u_char direction;
u_int ip_localaddr;
u_int ip_foreignaddr;
u_short tcp_localport;
u_short tcp_foreignport;
u_long snd_cwnd;
u_long snd_wnd;
u_long snd_ssthresh;
int conn_state;
u_int max_seg_size;
u_char in_fast_recovery;
u_char was_in_fast_recovery;
int smoothed_rtt;
u_char sack_enabled;
u_char in_slow_start;
STAILQ_ENTRY(pkt_node) nodes;
};
```

Listing 12

```
STAILQ_HEAD(pkthead, pkt_node) pkt_queue =
STAILQ_HEAD_INITIALIZER(pkt_queue);
```

Listing 13 initialises the previously declared pkt_queue struct so that it is ready to be used as a list. The STAILQ_INIT() macro can also be used to completely obliterate an existing list. Be aware though that any dynamically allocated memory associated with the nodes in the list will not be freed by calling STAILQ_INIT() on an existing list head. Therefore, before obliterating an existing list, make sure you iterate through the list and free all dynamically allocated memory first.

Listing 13

```
STAILQ_INIT(&pkt_queue);
```

Listing 14 appends a newly created pkt_node struct named pkt_node at the end of our queue.

Listing 14

```
STAILQ_INSERT_TAIL(&pkt_queue, pkt_node,
nodes);
```

Listing 15 grabs the node at the head of the list, storing the pointer to the node in pkt_node. The node is still left

in the list. If the list is empty, pkt_node will be assigned NULL.

Listing 15

```
pkt_node = STAILQ_FIRST(&pkt_queue)
```

Listing 16 removes the node at the head of the list. Removing the node simply disentangles it from the list structure, and does not free any dynamically allocated memory associated with the node.

Listing 16

```
STAILQ_REMOVE_HEAD(&pkt_queue, nodes);
```

B. Hash Tables

Hash tables are the other data structure we will discuss in detail. Hash tables are used when very fast, constant time lookups are required in time sensitive code. SIFTR uses a hash table to store the TCP connection packet counters. Hash tables are essentially an array, with the index of elements stored in the array calculated by "hashing" a key value. Ideally, 2 different keys should never hash to the same index. In practice, there is a very small probability of this occurring, and is referred to as a collision.

There are many different flavours of hash table and collision resolution. The FreeBSD hash table implementation [11] is known as a chaining hash, and is essentially an array of lists. If hashing a key causes a collision (an index containing a non-empty list), you simply append the node to the list located at the hashed index. Retrieving an element from the hash table requires the key to be hashed, and then searching the list found at the hash index to extract the correct element.

Listing 17 defines a new struct named flow_hash_node, which acts as the fundamental unit of data stored in our hash table. Each node in the table will contain an unsigned short counter for the number of flow packets observed, an unsigned char buffer holding the flow's hash key, and a link to the next node in the list. Each list in the hash table array is constructed in the same manner as previously described for lists/queues, except that FreeBSD hash tables use lists instead of tail queues. This means the LIST_* macros need to be used to operate on the lists.

Listing 17

```
struct flow_hash_node
{
u_short counter;
u_char key[12];
LIST_ENTRY(flow_hash_node) nodes;
};
```

Listing 18 defines a pointer to a struct named counter_hash which is the global reference to our hash table. This may look odd at first, as it appears to have nothing to do with an array and is simply a pointer to the head of a list. Remember though that arrays in C are simply a pointer to the beginning of a chunk of memory designated to hold some number of a particular type of object. We can therefore initialise our counter_hash pointer to point to any number of lists i.e. an array.

Listing 18

```
LIST_HEAD(counterhead, flow_hash_node)
*counter_hash;
```

Listing 19 initialises the previously declared counter_hash pointer to be an array of size SIFTR_EXPECTED_MAX_TCP_FLOWS.
Each position in the array will contain a LIST_HEAD(counterhead, flow_hash_node) struct, which forms the head of the list at each array index. The M_TEMP parameter simply specifies a textual name to be passed to the malloc() system call within the hashinit() function for debugging purposes. Finally, siftr_hashmask is an unsigned long which gets populated by hashinit() with a bitmask used to round hash indexes produced by the hash function down to be within the range of available array indexes.

Listing 19

```
counter_hash = hashinit(
SIFTR_EXPECTED_MAX_TCP_FLOWS, M_TEMP,
&siftr_hashmask);
```

Listing 20 obtains a pointer to the list head struct stored in the hash table at the index that "key" hashed to. The hash32_buf() function takes the key, key size and an initialisation vector as arguments, and returns the hashed value for the key as an unsigned int. Being an unsigned int, the value can be greater than the size of our hash table, which was initialised

to hold SIFTR_EXPECTED_MAX_TCP_FLOWS elements. We have to bitwise AND the hash value with the siftr_hashmask to ensure the hash value is within the range of the array dimensions i.e. between 0 and SIFTR_EXPECTED_MAX_TCP_FLOWS.

Some pointer arithmetic is then used on our hash table array pointer "counter_hash" to access the calculated hash index and assign it to counter_list. The counter_list variable can now be used as a normal list head struct pointer, and can be operated on using the LIST_* macros.

Listing 20

```
counter_list = (counter_hash+(hash32_buf(key, sizeof(key), 0) & siftr_hashmask));
```

Listing 21 destroys the previously initialised counter_hash hash table. It frees all memory dynamically allocated by hashinit(), but expects all lists within the hash table to be empty BEFORE calling it, and will cause a system panic if this is not the case i.e. you must loop through the hash table and empty every list before calling hashdestroy().

Listing 21

```
hashdestroy(counter_hash, M_TEMP, siftr_hashmask);
```

VI. Character Devices

One technique for transferring data from the kernel to user-space is to create a character device under the /dev device filesystem. Although we didn't end up using this in SIFTR, it may be useful in other instances.

As the final version of SIFTR doesn't create a character device, this section of the report refers to code snippets taken from an early prototype of SIFTR named siftrdev, which we have made available for download from [2].

Listing 22 declares a cdevsw struct (defined in /usr/src/sys/sys/conf.h) named tcpstats_cdevsw that defines the basic properties of our character device. Setting the .d_open .d_close and .d_read struct members to the locally defined dev_open dev_close and dev_read functions respectively sets up the 3 basic operations supported by our device i.e. opening, closing and reading. The dev_open() dev_close() and dev_read() functions will be called when the corresponding action is performed on the device.

Listing 22

```
static struct cdevsw tcpstats_cdevsw = {
.d_version = D_VERSION,
.d_open = dev_open,
.d_close = dev_close,
.d_read = dev_read,
.d_name = "tcpstats",
.d_flags = D_TTY,
};
```

The device is actually created during module initialisation using the make_dev() function, as shown in listing 23. The first argument is the tcpstats_cdevsw struct discussed previously. The second argument provides the dev_t device number identifier for the new device. The third and fourth arguments define the owner of the device, while the fifth defines the permissions. The final argument gives the device a name.

Listing 23

```
sdev = make_dev(&tcpstats_cdevsw, FLWACCT_MINOR, UID_ROOT, GID_WHEEL, 0600, "tcpstats");
```

Listings 24, 25 and 27 show the function prototypes required for a basic character device. It's possible to define other functions to customise behaviour by adding them to the cdevsw struct that is used to create the device.

The dev_open function will be called when a user opens the device to begin reading, and can be used to perform any initialisation in preparation for their read. In our prototype it just ensures no more than one user can open the device at a time.

Listing 24

```
int dev_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
```

The dev_read function should copy any necessary data from the kernel into user-space using the uiomove() function as shown in listing 26. uiomove takes three arguments: the buffer to copy from, the number of bytes to copy and a uio struct that defines the properties of the copy. For more information on uiomove, refer to its man page [12].

Listing 25

```
int dev_read(struct cdev *dev, struct uio *uio, int ioflag)
```

Listing 26

```
return uiomove(flwbuf, MIN(uio->uio_resid, strlen(flwbuf)), uio);
```

The dev_close function is called when a user that has the device open no longer needs it. It can perform any necessary cleanup, although in our prototype it just releases a lock, allowing other users to open the device.

Listing 27

```
int dev_close(struct cdev *dev, int fflag, int devtype, struct thread *td)
```

Finally, when the module is unloaded the device needs to be unregistered and removed from the system. Listing 28 destroys the tcpstats character device. It takes a single argument, being the pointer to the cdev struct returned by the make_dev function during the module's initialisation.

Listing 28

```
destroy_dev(sdev);
```

The easiest way to test the new device is to "cat" it e.g. "cat /dev/tcpstats". For more advanced access and logging, the standard C library can be used to write an application that reads data from the device. An example of a user-space C logging app has also been provided in the siftrdev code available from [2].

Another useful character device coding example can be found here [4].

VII. Threading and Associated Issues

Being a multi-threaded operating system, the FreeBSD kernel makes extensive use of threads to parallelise tasks where possible. Threaded programming is difficult at the best of times, and doing it in the kernel is certainly not for the feint-hearted.

A. Kernel Threads

Thread creation is performed using the kthread KPI [13]. Listing 29 creates a new kernel thread that will execute the function siftr_log_writer_thread() as its main. The second argument can be used to pass an arbitrary pointer to the new thread, but is not used in our case. The third argument can be used by the caller to obtain a proc struct containing information about the newly created thread. The RFNOWAIT flag, described in the rfork [14] documentation, specifies that the thread's parent process should not wait for the thread to return to it. The fifth parameter specifies the number of pages to make the new thread's stack, with 0 meaning use the system default. Finally, the sixth parameter gives the thread a name to identify it by in utilities such as "top -S".

The *curthread* thread struct pointer, defined in /usr/src/sys/sys/pcpu.h, can be used to obtain the details of the currently executing thread from within the thread itself.

Listing 29

```
kthread_create((void *)&siftr_log_writer_thread, NULL, NULL, RFNOWAIT, 0, "siftr_log_writer_thread");
```

Listing 30 terminates a kernel thread, returning 0 to the system indicating that the thread terminated without encountering any errors. Functions executing as the entry point (main) of a kernel thread should always use kthread_exit() to terminate instead of returning to their caller.

Listing 30

```
kthread_exit(0);
```

B. Glorious Sleep

Typically when writing a function that will execute as the entry point (main) of a kernel thread, an (almost) endless loop will be created to perform the required processing until some condition terminates the thread. If the loop was allowed to run at full speed, it would completely use all of the available processing resources and bring the system to a halt. Generally, you only need the thread to run every now and again, either based on some event occurring e.g. a new packet is ready for processing, or based on some required interval at which a task must be periodically performed.

The sleep [15] KPI provides the functions required to put a thread to sleep and to wake it up again.

Listing 31 puts the calling thread to sleep until it is explicitly woken up or until siftr_pkt_manager_thread_sleep_ticks/HZ amount of time has elapsed. The first parameter to tsleep() is an arbitrary pointer, used as an identifier for the subsequent call to wakeup() to know which thread to wake up.

The second parameter specifies the priority with which the sleeping thread should be checking for wake up calls. The third parameter specifies a string detailing the reason for the sleep, and is displayed by utilities like "top -S". The final parameter specifies the maximum sleep time before the thread is woken up.

By setting the maximum sleep time to 0, the thread will not wake up until a call to wakeup(&wait_for_pkt) is made. Setting it to a value greater than 0, for example 50, would cause the thread to wake up after $50/HZ$ seconds had elapsed if no call to wakeup() had been made since the thread went to sleep.

The HZ parameter refers to the kernel's tick rate, which can be obtained from the "kern.clockrate" syctl variable in user-space, or the "hz" variable in kernel space as defined in /usr/src/sys/sys/kernel.h. For example, if the tick rate was set to 1000HZ, then calling tsleep() with a sleep time of 50 would cause the thread to wakeup after 50/1000 i.e. 50ms had elapsed (assuming a call to wakeup() was not made).

Listing 31

```
tsleep(&wait_for_pkt, PWAIT, "pktwait",
siftr_pkt_manager_thread_sleep_ticks);
```

Listing 32 informs any sleeping threads waiting on the wait_for_pkt identifier to wakeup. Using a combination of tsleep() with the timeout argument set to 0 and wakeup() is an efficient means of implementing an event based processing chain. The processing thread can be put to sleep, and the data collection thread can wake the processing thread up as soon as data is available for processing. This technique is also preferable for power saving reasons. Every time the thread wakes up, regardless of whether there is work to do, the CPU is prevented from entering into a low power state.

However, event based processing performed in this way can be higher overhead than using a poll based mechanism if the events occur within a short period of one another. For SIFTR, it was found that a poll based mechanism produced significantly lower load on the system than an event based mechanism when high packet rates are passing through the network stack.

Listing 32

```
wakeup(&wait_for_pkt);
```

C. Mutexes

One of the major difficulties with threaded programming is shared resources. When two threads can perform their functions completely independent of one another, and with completely independent data stores, everything will behave itself. However, problems arise if the two threads need to share information between them. Concurrent access to the same portion of memory causes systems to crash, and so access to shared memory has to be controlled so that only one thread can have exclusive access at any one time.

Mutexes [16] are the standard way of controlling access to shared resources in a threaded environment. The basic idea is that before running portions of code that access shared resources, the code must first "acquire" or "lock" the mutex. Once the mutex is acquired, the thread that has acquired the mutex can be sure that it is the only one that will be accessing the shared resource until it releases the mutex. If another thread attempts to acquire the mutex whilst the mutex is locked by another thread, the acquiring thread will wait until the mutex is released before it continues executing code.

The SIFTR code uses mutexes to control access to the data structures that are used to pass messages between the various threads in operation. When a thread is adding, reading or removing data from a shared data structure, a mutex is acquired prior to the operation, and released afterwards.

Listing 33 declares a new mutex struct named siftr_pkt_queue_mtx. You must declare a mutex before you initialise it, and must initialise it before you use it.

Listing 33

```
static struct mtx siftr_pkt_queue_mtx;
```

Listing 34 initialises our previously declared siftr_pkt_queue_mtx mutex struct with name set to siftr_pkt_queue_mtx. Setting the third parameter to NULL causes kernel debugging code to identify the mutex by its name. MTX_DEF mutexes are the most typical type of mutex, and should be used in most instances over MTX_SPIN mutexes (unless you know what you are doing and know you require a spin mutex).

Listing 34

```
mtx_init(&siftr_pkt_queue_mtx, "siftr_pkt_queue_mtx",
NULL, MTX_DEF);
```

Listing 35 lists the three functions related to mutex

locking. mtx_lock() will not return until it has managed to successfully lock the siftr_pkt_queue_mtx mutex. This means that if the mutex is already locked by another thread for example, the call the mtx_lock() will block (cause the thread to sleep) until the mutex becomes available. If your code is not time critical and can afford to block waiting for the mutex, mtx_lock() is the function for you.

mtx_trylock() will also attempt to acquire the siftr_pkt_queue_mtx, but will return immediately if the mutex is already locked. It returns 0 to indicate failure to acquire the mutex, and non-zero if it successfully acquired the mutex. This provides an appropriate way of acquiring a mutex only if it is available, in code that cannot afford to block. In SIFTR for example, the PFIL hook function we install to intercept packets cannot block at all, because the PFIL code in the kernel holds a non-sleepable mutex. We use mtx_trylock() in the hook function to ensure we never block waiting for the mutex to become available.

mtx_unlock() is called if a successful call to mtx_lock() or mtx_trylock() was made previously.

Listing 35

```
mtx_lock(&siftr_pkt_queue_mtx);
mtx_trylock(&siftr_pkt_queue_mtx)
mtx_unlock(&siftr_pkt_queue_mtx);
```

Finally, listing 36 destroys the previously initialised siftr_pkt_queue_mtx mutex.

Listing 36

```
mtx_destroy(&siftr_pkt_queue_mtx);
```

VIII. File Writing

File writing is something that is very commonly performed and easily achieved in user-space applications. However, manipulating files in kernel space turned out to be far more complicated than we expected it to be. Our initial attempt lead us down the path of attempting to open a file descriptor from within the kernel. After finally figuring out how this was done, we learnt that file descriptors are a per kernel thread entity, and cannot be used between different kernel threads i.e. opening the file descriptor in the module's init() function, writing to the descriptor in the PFIL hook and closing the descriptor in the module's deinit() function does not work, as each function is run by a different kernel thread.

We finally stumbled upon some code written by a FreeBSD kernel hacker that provided the functionality we required. The kernio [17] functions written by Pawel Jakub Dawidek allow us to open, write and close files from within the kernel, and work across different kernel threads. The kio interface is almost identical to the open(), write() and close() syscall functions available in user-space, except that vnode structures are used instead of file descriptors.

On a side note, Pawel informed us that the kernio code will be available in the upcoming FreeBSD 7.x series kernel source in the file sys/compat/opensolaris/kern/opensolaris_kobj.c.

Listing 37 opens the file located at the path stored in the siftr_logfile string for writing, creating it with mode 0644 if it does not exist already. If the file does exist, all writes to the file will be appended. A vnode that references the requested file is returned and stored as siftr_vnode for later use.

Listing 37

```
siftr_vnode = kio_open(siftr_logfile, O_CREAT |
O_WRONLY | O_APPEND, 0644);
```

Listing 38 writes *index* bytes from the *log_writer_msg_buf* pointer into the file referenced by *siftr_vnode*.

Listing 38

```
kio_write(siftr_vnode, log_writer_msg_buf, index);
```

Finally, listing 39 closes the vnode previously opened with kio_open().

Listing 39

```
kio_close(siftr_vnode);
```

An example kernel module named filewriter that shows off multi-threaded log file writing can be found here [2].

IX. Debugging

Things will inevitably go wrong, and you are often going to have to go to some lengths to figure out why before you can fix them for anything but the most trivial of bugs.

Debugging applications in user-space has gradually become easier with the availability of powerful graphical debugging tools e.g. DDD [18]. Kernel-space debugging

is regarded as something of a black art, and we hope to provide some basic pointers to get you started on the road to kernel debugging enlightenment.

For logic bugs, there are kernel versions of the user-space family of printf() style functions. Strings passed into printf() will be written to syslog, while uprintf() will write to the user's current terminal. Liberal (and temporary) use of these functions can be useful to track the value of key variables. However they will provide little help tracking down more serious errors.

Bugs that would cause a segfault in a user-space application (such as attempting to de-reference a NULL pointer) will often cause a kernel panic when they occur in kernel code. When a panic occurs an error message will briefly appear on tty0 before the machine resets itself.

Although difficult, tracking down the cause of the panic can be aided by configuring the kernel to dump a copy of the kernel state to disk before the reboot. This allows detailed analysis of the crash data to be performed after the reboot has occured.

Figure 40 contains the two lines that should be added to /etc/rc.conf to enable kernel dumping. The dumpdev option indicates the drive the state should be saved to before the reboot, which should generally be set to the partition that is used as a swap drive.

Listing 40

```
dumpdev=/dev/ad0s2b
dumpdir=/var/crash
```

As FreeBSD boots after the panic, it will save any dump data it detects on the dump device to the directory specified by the dumpdir option.

A short text log file will be saved along with the dump file with some basic information on the panic. Detailed information (such as the stacktrace leading up to the panic) can be obtained by opening the dump file with kgdb, as shown in listing 41. Note that the KERNCONF directory should be replaced with the name of the kernel that was running at the time, as indicated by "uname -i".

Listing 41

```
kgdb /usr/obj/usr/src/sys/KERNCONF/kernel.debug
/var/crash/vmcore.1
```

The Debugging Kernel Problems tutorial by Greg Lehey [19] provides a comprehensive guide to debugging issues in the FreeBSD kernel.

X. Where Can I Get Help?

The FreeBSD hackers mailing list was an invaluable source of information for us. FreeBSD runs a large number of mailing lists [20], many being related to a specific aspect of FreeBSD which may be more suited to your problem. Anyone can subscribe and there are many knowledgeable people reading the lists.

When asking kernel related questions, keep them specific and provide plenty of supporting information and code. The people with the experience to answer your questions are more likely to help out if they can see you've tried a few things and aren't expecting to have the solution created for you.

XI. Acknowledgements

This work was completed as part of a project funded by the Cisco University Research Program (URP).

The authors would also like to acknowledge the help of the FreeBSD community, namely Pawel Jakub Dawidek for creating the kernio code, and the users on the freebsd-hackers mailing list for the help and support they provided us during the creation of the SIFTR software.

References

[1] "The NewTCP Project," June 2007, http://caia.swin.edu.au/urp/newtcp.

[2] "NewTCP project tools," June 2007, http://caia.swin.edu.au/urp/newtcp/tools.html.

[3] FreeBSD Hypertext Man Pages, "KLD," June 2007, http://www.freebsd.org/cgi/man.cgi?query=kld&sektion=4.

[4] "FreeBSD Device Driver Template/Skeleton (Kernel Module)," June 2007, http://www.captain.at/howto-freebsd-device-driver-template-skeleton.php.

[5] FreeBSD Hypertext Man Pages, "SYSCTL," June 2007, http://www.freebsd.org/cgi/man.cgi?query=sysctl&sektion=9.

[6] ——, "PFIL," June 2007, http://www.freebsd.org/cgi/man.cgi?query=pfil&sektion=9.

[7] Murat Balaban, "Hooking Filters for Fun and Profit: PFIL_HOOKS," June 2007, http://www.enderunix.org/docs/en/pfil_hook.html.

[8] Stephanie Wehner, "Fun and Games with FreeBSD Kernel Modules," June 2007, http://www.itsx.com/hal2001/fbsdfun.html.

[9] J. Kong, *Designing BSD rootkits: an introduction to kernel hacking*. No Starch Press, Inc. (ISBN: 1593271425), 2007.

[10] FreeBSD Hypertext Man Pages, "QUEUE," June 2007, http://www.freebsd.org/cgi/man.cgi?query=queue&sektion=9.

[11] ——, "HASHINIT," June 2007, http://www.freebsd.org/cgi/man.cgi?query=hashinit&sektion=9.

[12] ——, "UIOMOVE," June 2007, http://www.freebsd.org/cgi/man.cgi?query=uiomove&sektion=9.

[13] ——, "KTHREAD," June 2007, http://www.freebsd.org/cgi/man.cgi?query=kthread&sektion=9.

[14] ——, "RFORK," June 2007, http://www.freebsd.org/cgi/man.cgi?query=rfork&sektion=2.

[15] ——, "SLEEP," June 2007, http://www.freebsd.org/cgi/man.cgi?query=sleep&sektion=9.

[16] ——, “MUTEX,” June 2007, http://www.freebsd.org/cgi/man.cgi?query=mutex&sektion=9.
[17] Pawel Jakub Dawidek, “kernio,” June 2007, http://people.freebsd.org/~pjd/misc/kernio/.
[18] “DDD - Data Display Debugger,” June 2007, http://www.gnu.org/software/ddd/.
[19] Greg Lehey, “Debugging Kernel Problems,” May 2006, http://www.lemis.com/grog/Papers/Debug-tutorial/tutorial.pdf.
[20] The FreeBSD Project, “lists.freebsd.org Mailing Lists,” June 2007, http://lists.freebsd.org/.